**Перетворення типів**

Перетворення типів у JavaScript - це процес зміни типу даних з одного у інший, наприклад, з рядка в число або з числа в булеве значення. Ми використовуємо перетворення типів для обробки та маніпуляції даними різних типів, щоб вони відповідали очікуваному формату операцій або функцій.

**Явне перетворення типів**

**З використанням конструкторів:**

JavaScript надає вбудовані конструктори, такі як Number(), String(), і Boolean(), які можна використовувати для явного перетворення типів.

const strToNum = Number("456"); // "456" стає числом 456

const boolToStr = String(true); // true стає рядком "true"

const numToBool = Boolean(0); // 0 стає булевим false, оскільки 0 вважається "falsy" значенням

За допомогою метода toString() та функцій parseInt(), parseFloat():

const num = 12345.6789;

const numToStr = num.toString(); // Конвертує число в рядок "12345.6789"

const strToInt = parseInt("100px"); // Конвертує рядок в ціле число 100, ігноруючи нечислові символи

const strToFloat = parseFloat("3.14someText"); // Конвертує рядок в число з плаваючою комою 3.14

### Неявне перетворення типів

JavaScript також виконує неявне перетворення типів, коли це необхідно для операцій.

const result = "3" + 2; // JavaScript перетворює число 2 в рядок і конкатенує його з "3", результат "32"

const comparison = "5" == 5; // JavaScript перетворює рядок "5" в число 5 перед порівнянням, результат true

### Швидке перетворення типів

Для швидкого перетворення типів існують короткі вирази:

const strToNumQuick = +"42"; // Швидке перетворення рядка "42" в число за допомогою унарного оператора +

const numToStrQuick = 123 + ""; // Швидке перетворення числа 123 в рядок за допомогою конкатенації з порожнім рядком

const boolToNum = +true; // Швидке перетворення булевого true в число 1

### Перетворення та порівняння

Під час порівняння значень різних типів JavaScript може виконувати неявне перетворення типів, що може призвести до неочікуваних результатів.

const compare = 0 == false; // true, оскільки 0 і false є "falsy" значеннями

Знання та розуміння механізмів перетворення типів є важливим для ефективної роботи з даними в JavaScript, оскільки дозволяє уникнути помилок та неправильної поведінки програми.

## Управління потоком виконання

Управління потоком виконання програми в JavaScript здійснюється за допомогою умовних конструкцій, таких як if, if ... else, if ... else if ... else, та switch ... case. Ці конструкції дозволяють програмі виконувати різні блоки коду залежно від заданих умов.

### if {}

Конструкція if використовується для виконання коду, якщо задана умова істинна.

if (condition) {

// Код, що виконується, якщо умова істинна

}

Приклад:

var age = 20;

if (age >= 18) {

console.log("Ви можете голосувати.");

}

### if ... else

if else додає альтернативну дію, яка виконується, якщо умова хибна.

if (condition) {

// Код, що виконується, якщо умова істинна

} else {

// Код, що виконується, якщо умова хибна

}

Приклад:

var score = 75;

if (score >= 50) {

console.log("Ви пройшли тест.");

} else {

console.log("Ви не пройшли тест.");

}

### if ... else if ... else

Цей варіант використовується для створення кількох умовних гілок.

if (condition1) {

// Код, що виконується, якщо condition1 істинна

} else if (condition2) {

// Код, що виконується, якщо condition2 істинна

} else {

// Код, що виконується, якщо обидві умови хибні

}

Приклад:

var score = 85;

if (score >= 90) {

console.log("Відмінно");

} else if (score >= 75) {

console.log("Добре");

} else {

console.log("Задовільно");

}

### switch ... case

switch використовується для виконання різних дій на основі різних умов.

switch (expression) {

case value1:

// Виконується, якщо expression дорівнює value1

break;

case value2:

// Виконується, якщо expression дорівнює value2

break;

default:

// Виконується, якщо жоден з вищезазначених випадків не відбувся

}

Приклад:  
var day = 2;

switch (day) {

case 1:

console.log("Понеділок");

break;

case 2:

console.log("Вівторок");

break;

case 3:

console.log("Середа");

break;

default:

console.log("Інший день тижня");

}

Ці конструкції є невід'ємною частиною JavaScript для управління потоком програми, дозволяючи створювати гнучкі та ефективні рішення.

## Умовний (Тернарний) оператор та Оператор кома

### Умовний (Тернарний) оператор

Умовний (тернарний) оператор у JavaScript є компактною альтернативою стандартній умовній конструкції if ... else. Він названий "тернарним", оскільки включає три частини: умову, вираз, що виконується, якщо умова істинна, і вираз, що виконується, якщо умова хибна. Синтаксис тернарного оператора виглядає так: умова ? вираз1 : вираз2;.

### Приклади використання тернарного оператора:

**Задання змінної в залежності від умови:**

var age = 20;

var canVote = (age >= 18) ? "так" : "ні";

console.log("Чи може особа голосувати? " + canVote); // Виведе: "Чи може особа голосувати? так"

У цьому прикладі в залежності від того, чи задовольняє змінна age умові бути не менше 18, змінній canVote присвоюється відповідне значення "так" або "ні".

**Використання тернарного оператора для виведення повідомлень:**

var score = 75;

score >= 50 ? console.log("Ви пройшли") : console.log("Ви не пройшли");

Тут, залежно від результату порівняння score з 50, в консоль виводиться одне з двох повідомлень: "Ви пройшли" або "Ви не пройшли".

**Вкладені тернарні оператори для більш складних умов:**

const marks = 85;

let grade;

switch (true) {

case marks >= 90:

grade = 'A';

break;

case marks >= 80:

grade = 'B';

break;

case marks >= 70:

grade = 'C';

break;

case marks >= 60:

grade = 'D';

break;

default:

grade = 'F';

}

console.log("Оцінка: " + grade); // Виведе: "Оцінка: B"

Тут використовується вкладений тернарний оператор для визначення оцінки за балами. Це дозволяє замінити більш громіздку конструкцію if ... else if ... else.

Тернарний оператор дозволяє зробити код більш лаконічним і зручним для читання, особливо коли потрібно просто вибрати одне з двох значень на основі умови. Однак важливо не перевантажувати код вкладеними тернарними операторами, оскільки це може ускладнити його розуміння.

Error

### Оператор кома

Оператор кома в JavaScript дозволяє нам виконати кілька виразів, розділених комою, в одному операторі. В результаті виконання такої конструкції повертається значення останнього виразу. Це зручно використовувати для виконання кількох дій в одному рядку коду або в інструкціях управління, де очікується один вираз, наприклад, в умовах циклів.

**Оператор кома використовується для:**

* Мінімізації коду шляхом виконання декількох операцій в одному місці без необхідності розділення їх на декілька рядків.
* Ініціалізації або оновлення кількох змінних в циклі.

Приклади:

**Ініціалізація кількох змінних:**

var a = 1, b = 2, c = 3; // Встановлює значення для a, b, і c

**Використання в циклі for:**

for (var i = 0, j = 10; i <= 10; i++, j--) {

console.log(i, j); // Виводить пари чисел: 0 10, 1 9, ..., 10 0

}

Тут оператор кома використовується для одночасної ініціалізації i і j на початку циклу та для їх оновлення з кожною ітерацією.

**Оцінка кількох виразів:**

var x, y, z;

x = (y = 5, z = 10); // y стає 5, z стає 10, і x стає 10

В цьому прикладі оператор кома дозволяє присвоїти y значення 5 і z значення 10. Значенням x стає результат останнього виразу, тобто 10.

Оператор кома є корисним інструментом для спрощення коду та здійснення кількох дій в одному виразі, але важливо використовувати його обережно, щоб не ускладнити читання та розуміння коду.

## Цикли

У JavaScript цикли використовуються для повторення блоку коду кілька разів. Кожен тип циклу має своє призначення та сценарій використання.

Цикл for є стандартним способом повторення операцій, який дає можливість ініціалізувати змінну, перевіряти умову, та змінювати змінну на кожній ітерації.

for (var i = 0; i < 5; i++) {

console.log(i); // Виводить числа від 0 до 4

}

Цикл for...in перебирає всі перелічувані властивості об'єкта.

var obj = {a: 1, b: 2, c: 3};

for (var prop in obj) {

console.log(prop + ": " + obj[prop]); // Виводить "a: 1", "b: 2", "c: 3"

}

Цикл for...of дозволяє перебирати ітеровані об'єкти (включно з масивами, рядками, і Map/Set).

var arr = [10, 20, 30];

for (var value of arr) {

console.log(value); // Виводить 10, 20, 30

}

Цикл while виконує блок коду, допоки задана умова істинна.

var i = 0;

while (i < 5) {

console.log(i); // Виводить числа від 0 до 4

i++;

}

Цикл do...while схожий на while, але гарантує, що тіло циклу виконається хоча б один раз.

var i = 0;

do {

console.log(i); // Виводить числа від 0 до 4

i++;

} while (i < 5);

### Оператори для роботи з циклами

* **break** завершує виконання поточного циклу, переходячи до коду, що йде після циклу.

for (var i = 0; i < 10; i++) {

if (i === 5) {

break; // Виходимо з циклу, коли i дорівнює 5

}

console.log(i); // Виводить числа від 0 до 4

}

* **continue** пропускає поточну ітерацію циклу і переходить до наступної ітерації.

for (var i = 0; i < 10; i++) {

if (i === 5) {

continue; // Пропускаємо 5

}

console.log(i); // Виводить числа від 0 до 4 та від 6 до 9

}

* **label** дає можливість ідентифікувати цикл або блок коду, щоб потім використовувати з break або continue.

outerLoop: for (var i = 0; i < 3; i++) {

for (var j = 0; j < 3; j++) {

if (i === j) {

continue outerLoop;

}

console.log("i = " + i + ", j = " + j);

}

}

Кожен з цих циклів та операторів має своє унікальне застосування та може бути корисним у різних сценаріях програмування для управління потоком виконання програми.

## Функції/Callback функції

### Функції

**Функція** в програмуванні - це відокремлений блок програмного коду, який розробляється для виконання певної задачі або обчислення. Вона може приймати дані на вході (аргументи), обробляти їх та повертати результат.

Функції створюються для того, щоб код, який виконує певну операцію, міг легко викликатися з різних місць програми без необхідності його дублювання.

У JavaScript функції є основними будівельними блоками програм, що дозволяють упаковувати блоки коду для виконання конкретних задач. Функції можна створювати різними способами, і вони дозволяють працювати з даними, переданими як аргументи, та повертати результати.

### Function Declaration (Оголошення функції)

Оголошення функції (або Function Declaration) визначає функцію з вказаним іменем.

function sum(a, b) {

return a + b;

}

console.log(sum(1, 2)); // Виведе: 3

Оголошення функції "піднімається" (hoisted), що означає можливість викликати функцію до її оголошення у коді.

### Function Expression (Функціональний вираз)

Функціональний вираз (або Function Expression) означає присвоєння функції змінній. Функція може бути анонімною або мати ім'я.

var multiply = function(a, b) {

return a \* b;

};

console.log(multiply(3, 4)); // Виведе: 12

Функціональний вираз не піднімається, тобто функцію не можна викликати перед її визначенням у коді.

### Arguments (Аргументи)

Ключове слово arguments є об'єктом, схожим на масив, доступним всередині функцій, що містить аргументи, передані функції.

function printArguments() {

for (var i = 0; i < arguments.length; i++) {

console.log(arguments[i]);

}

}

printArguments('Hello', 'World', '!'); // Виведе: "Hello", "World", "!"

Ця можливість дозволяє функціям працювати з не визначеною заздалегідь кількістю аргументів.

Error

**Callback функції**

Callback функції в програмуванні — це функції, які передаються як аргументи іншим функціям.

Ми використовуємо callback-функції для того, щоб забезпечити можливість виклику певного коду після завершення виконання іншого коду, що дозволяє нам створювати асинхронний код, наприклад, при роботі з запитами до сервера або обробці подій.

**Приклади використання callback-функцій:**

**Основний приклад:**

function doHomework(subject, callback) {

console.log(`Починаємо робити домашнє завдання з ${subject}.`);

callback();

}

doHomework('математики', function() {

console.log('Домашнє завдання завершено!');

});

У цьому прикладі, функція doHomework приймає назву предмету та callback-функцію як аргументи. Після виводу повідомлення про початок виконання домашнього завдання, вона викликає передану callback-функцію, що повідомляє про завершення.

**Асинхронний приклад з setTimeout:**

function washDishes(callback) {

console.log('Початок миття посуду.');

setTimeout(() => {

// Припустимо, що миття посуду триває 3 секунди

callback();

}, 3000);

}

washDishes(function() {

console.log('Посуд вимито!');

});

У цьому випадку, washDishes імітує асинхронну операцію миття посуду за допомогою setTimeout. Callback-функція викликається після завершення цієї операції.

### Переваги використання callback-функцій:

1. **Асинхронність:** Callback-функції дозволяють JavaScript продовжувати виконання коду, не чекаючи завершення попереднього завдання, що є особливо корисним при виконанні операцій, які можуть зайняти багато часу, наприклад, запитів до сервера.
2. **Гнучкість:** Вони дозволяють створювати більш модульний і гнучкий код, розділяючи логіку програми на менші частини.
3. **Контроль потоку:** Callback-функції можуть допомогти керувати послідовністю виконання операцій, особливо коли порядок виконання має значення.

Використання callback-функцій є ключовим аспектом асинхронного програмування в JavaScript, проте важливо звернути увагу на потенційні проблеми, такі як "callback hell" або "pyramid of doom", коли велика кількість вкладених callback-функцій робить код важким для розуміння та підтримки. Для вирішення цих проблем можна використовувати Проміси та async/await (будемо розглядати пізніше).